Cmpe300
Analysis of Algorithms
Tunga Güngör

Student:Emre Boran
ID:2013300075

Submitted Person:
Emrah Budur

Query Expansion
MPI Application Project
11.05.2019

Introduction

When we type a word to a typical search engine, the result is not only the query terms but also some other terms that are relevant to our query terms. For example, when we type ‘boğaziçi’, the result can include relevant words such as odtü, university, education, the place which is called boğaziçi, boğaziçi elektrik, etc. The method is called **query expansion**. One way to make fast query expansion to use parallel programming.

In this project, I have used words’ embeddings to find relevant words. A **word embedding** is a fixed-sized vector of real numbers that represents the position of a specific word in an high dimensional space. It is the representation of words according to dimensioans. We can find similarity of two words’ embedding by using cosine similarity.

Also, I have used c**osine similarity** which is a measure of similarity between two non-zero vector.

To make faster this setup, I have used parallel programming in this project. How many processors the program takes is given when running. Note that if P+1 processors is given, it means we have 1 master and P slave processors which is also called as node. Whenever I write P, I mean slave nodes.

Program Interface

You can execute the following commands to compile and run the sample application. I have written this program in ubuntu. When user wants to exit, it should write ‘exit’ or ‘EXIT’ for query word.

NUM_OF_PROCESSORS = P + 1

```
 * Windows (MPICH2)
 gcc -L"C:\Program Files (x86)\MPICH2\lib" -I"C:\Program Files (x86)\MPICH2\include"
mpi_project.c -lmpi -o mpi_project.exe
mpiexec -n NUM_OF_PROCESSORS ./mpi_project.exe


* Unix/Max (OpenMPI)

mpicc mpi_project.c -o mpi_project.o
mpirun -np NUM_OF_PROCESSORS ./mpi_project.o

* Ubuntu
mpicc -g mpi_project.c -o mpi_project

mpiexec -n NUM_OF_PROCESSORS ./mpi_project
```

Program Execution

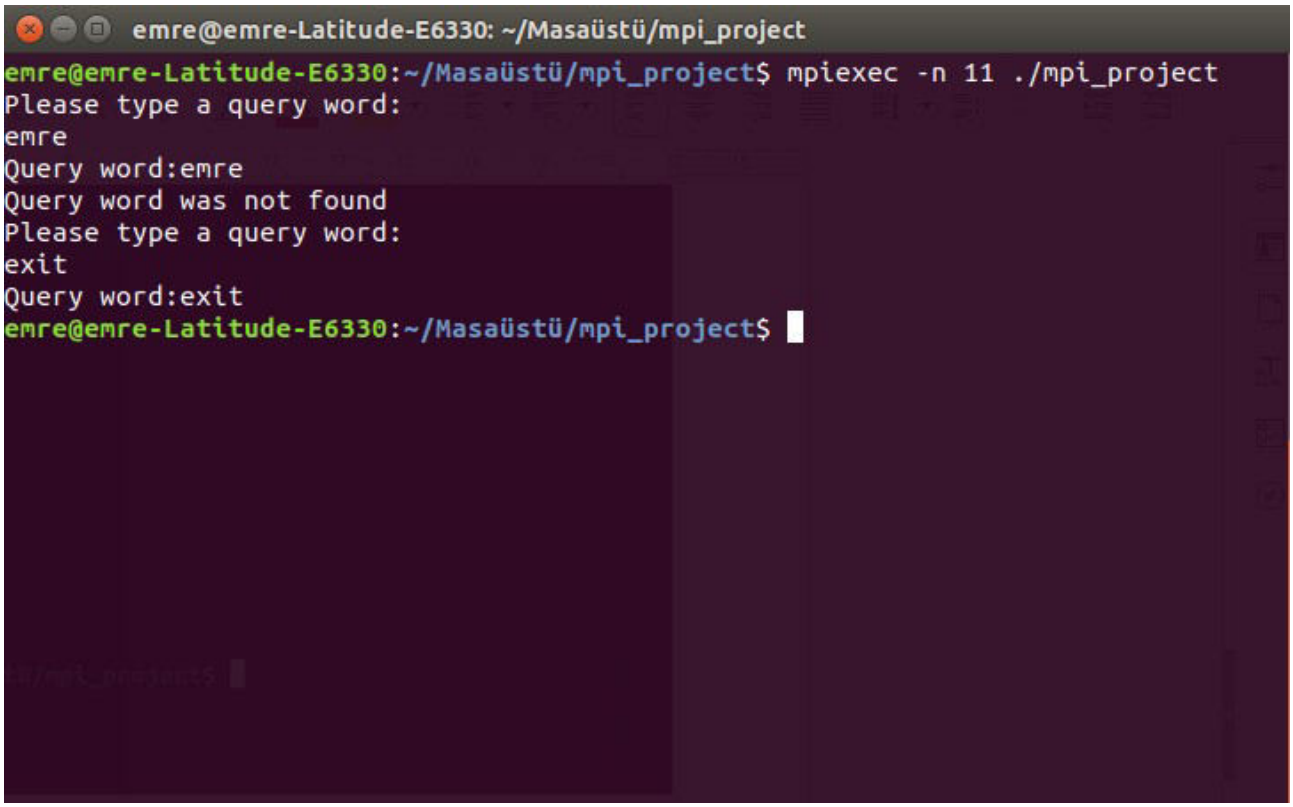
After running program according to program interface, program ask typing a query word.

```
emre@emre-Latitude-E6330: ~/Masaüstü/mpi_project
roject
emre@emre-Latitude-E6330:~/Masaüstü/mpi_project$ mpiexec -n 11 ./mpi_project
Please type a query word:
```

After user write a query word and enter program writes most relevant P words.

```
emre@emre-Latitude-E6330: ~/Masaüstü/mpi_project
roject
emre@emre-Latitude-E6330:~/Masaüstü/mpi_project$ mpiexec -n 11 ./mpi_project
Please type a query word:
boğaziçi
Query word:boğaziçi
TOP 10 RESULTS:
boğaziçi: 1.000000
rumelihisarı: 0.644971
marmara: 0.639540
odtü: 0.635023
istanbul: 0.630798
ayazağa: 0.623934
boğaz: 0.622000
bilkent: 0.612875
ortaköy: 0.612030
iskelesi: 0.601631
Please type a query word:
exit
Query word:exit
emre@emre-Latitude-E6330:~/Masaüstü/mpi_project$
```

Then, program ask new query word. It continues until user write ‘exit’ or ‘EXIT’ as a query. After that program ends. If the user write a word which is not included input file that is explained next section program writes: Query word was not found.



Input and Output

After running program, as described in program execution user should type a word. It is the input from user. Another input is the file that includes words embeddings. The file have 1000 row and 300 dimension in our input. Every rows starts with the word and 300 float number between -1 and 1 in order to represent dimensions of word vector. Output is the most relevant P word.

The bigger input file, the more relevant words can be found. For example, in the above screenshots there will be a words higher relevant than rumelihisarı to boğaziçi but there is no such a word our input file.

Program Structure

All implementation is the same with pseudocode given in project description. Also I have commented every detail of program in source code so I will screanshots given appendix section to describe program details. However, I want to clearify some parts here.

i. Function runSlaveNode

RunSlaveNode function is returning an array of most relevant P words and index. I have found the most relevant P words and index with a way in which I have used a part of insertion sort.

I have created three array have size P. These are bestPscore, sortedwords and bestPindex. When looking to find most relevant P words in the slave, arrays are filled in descending order by using insertion sort's insertion teqnique.

ii. Function runMasterNode

RunMasterNode manages all slave processors. After all slave nodes returns their P most relevant words to master node, master node should find P most relavant word among P*P words.

First, I have created two array that has size P*P named bestPscore and words and three array that has size P named outputPscore, outputWords and ptrScore.

bestPscore and words holds words accordingly slave nodes, which means that if third slave nodes return their part of P words and scores and if P = 10, the words and scores is inserted between bestPscore[20:29], words[20:29].

Assume that P = 10. ptrScore is [0,10 ... 80,90] when starting. It holds the starting point of subarray that comes from slaves.

Two nested loop solves finding the most relevant P words and scores. When outer loop executes one, it is found the most relevant words among remaining. When inner loop complete execution P times, it is found most relevant words among P words. It looks top element of subarrays by looking indexes written in ptrScore.

Examples

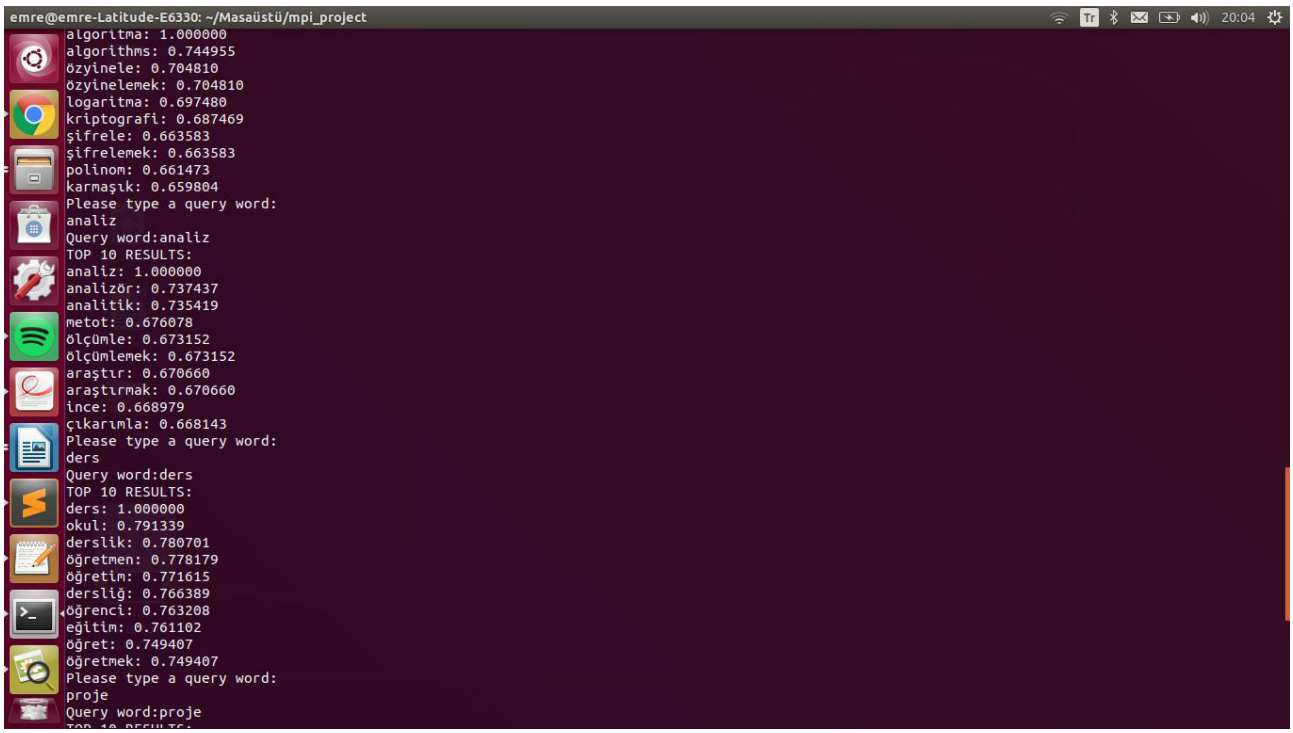
```
emre@emre-Latitude-E6330:~/Masaüstü/dersler/cmpe300/MPI/CMPE3
/CMPE300_Spring_2019_MPI_PS$ mpiexec -n 11 ./mpi_project
Please type a query word:
boğaziçi
Query word:boğaziçi
TOP 10 RESULTS:
boğaziçi: 1.000000
rumelihisarı: 0.644971
marmara: 0.639540
odtü: 0.635023
istanbul: 0.630798
ayazağa: 0.623934
boğaz: 0.622000
bilkent: 0.612875
ortaköy: 0.612030
iskelesi: 0.601631
Please type a query word:
marmara
Query word:marmara
TOP 10 RESULTS:
marmara: 1.000000
yalova: 0.677468
gebze: 0.676491
karadeniz: 0.669564
marmaris: 0.666930
boğaziçi: 0.639540
pendik: 0.627666
büyükçekmece: 0.617254
akdeniz: 0.615086
maltepe: 0.612199
Please type a query word:
exit
Query word:exit
```

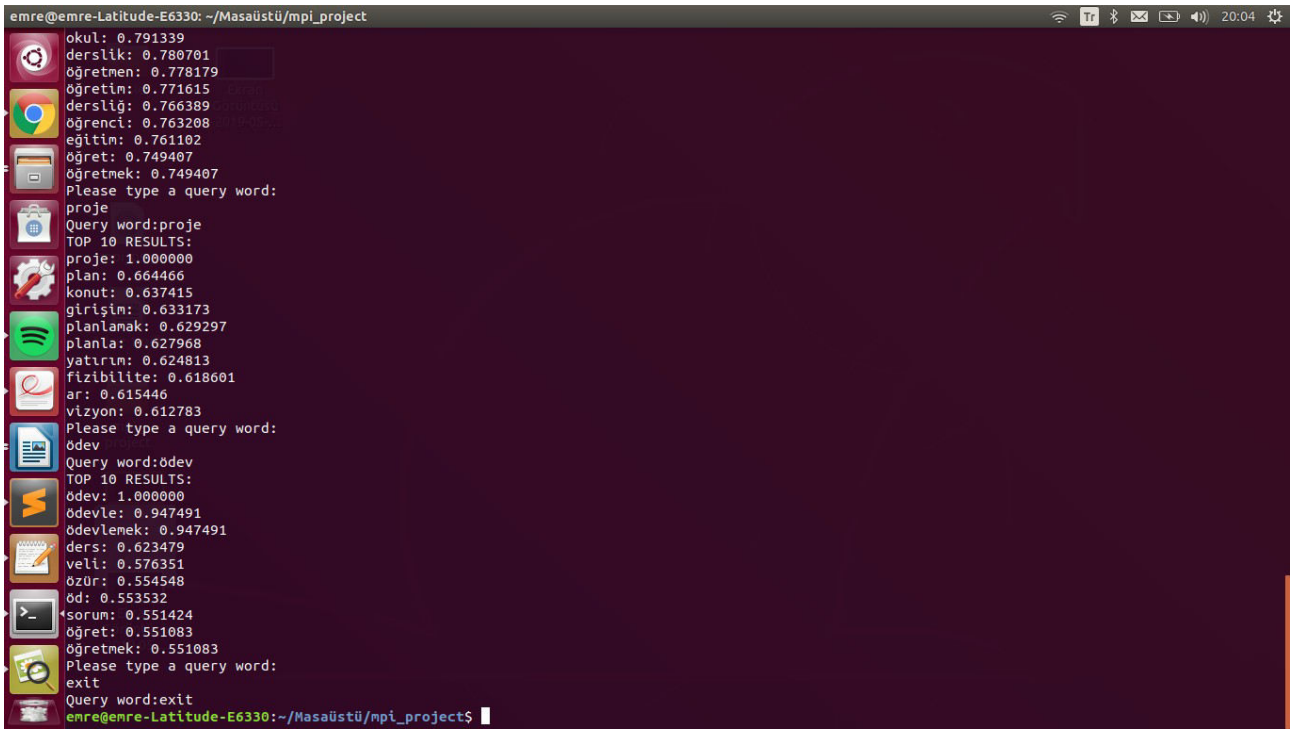
In this example, query similarity results boğaziçi and marmara are matching. Both of them 0.639549. Also, they are most similar,1.00000, with itself.

Examples for bonus part

```
emre@emre-Latitude-E6330: ~/Masaüstü/mpi_project
emre@emre-Latitude-E6330:~/Masaüstü/mpi_project$ mpicc -g mpi_project.c -o mpi_project
emre@emre-Latitude-E6330:~/Masaüstü/mpi_project$ mpiexec -n 11 ./mpi_project
Please type a query word:
boğaziçi
Query word:boğaziçi
TOP 10 RESULTS:
boğaziçi: 1.000000
rumelihisarı: 0.644971
marmara: 0.639540
odtü: 0.635023
istanbul: 0.630798
ayazağa: 0.623934
boğaz: 0.622000
bilkent: 0.612875
ortaköy: 0.612030
iskelesi: 0.601631
Please type a query word:
üniversite
Query word:üniversite
TOP 10 RESULTS:
üniversite: 1.000000
üniversitesi: 0.924389
fakülte: 0.838503
rektör: 0.784243
öğrenci: 0.770045
kolej: 0.732595
kampüs: 0.718077
burs: 0.716351
mezun: 0.708246
dekan: 0.704746
Please type a query word:
bilgisayar
Query word:bilgisayar
TOP 10 RESULTS:
bilgisayar: 1.000000
dizüstü: 0.761838
yazılım: 0.740554
cihaz: 0.714108
çip: 0.704136
aygıt: 0.702887
```

```
emre@emre-Latitude-E6330: ~/Masaüstü/mpi_project
cihaz: 0.714108
çip: 0.704136
aygıt: 0.702887
harddisk: 0.697283
masaüst: 0.697266
masaüstü: 0.691609
bellek: 0.673513
Please type a query word:
mühendis
Query word:mühendis
TOP 10 RESULTS:
mühendis: 1.000000
başmühendis: 0.753364
teknisyen: 0.711157
mühendishane: 0.675722
tekniker: 0.663020
mimar: 0.620890
bilim: 0.614011
jeofizik: 0.608447
mucit: 0.605955
uzman: 0.605567
Please type a query word:
bölüm
Query word:bölüm
TOP 10 RESULTS:
bölüm: 1.000000
bölü: 0.939717
bölümle: 0.836784
bölümlemek: 0.836784
sezon: 0.649102
dizi: 0.630105
işbölüm: 0.597805
işbölümü: 0.597805
jenerik: 0.579716
altbölüm: 0.578744
Please type a query word:
algoritma
Query word:algoritma
TOP 10 RESULTS:
algoritma: 1.000000
algorithms: 0.744955
özyinele: 0.704810
```





Improvements and Extensions

In runMasterNode, when finding the most P relevant word, binary heap could be used instead of inner loop. Now, complexity is O(p*p) in this part. With binary heap, complexity could be O(plogp).

Difficulties Encountered

I had difficulties with the MPI environment installation. Also operations including pointers takes some time.

Conclusion

With this project I have learnt a lot of issues about the parallel programming and mpi environment. I think the program works fine. But it need some improvements as I mentioned in the improvement section.

Appendices

mpi_project.c

```c
/*
Student Name: Emre Boran
Student Number: 2013400210
Compile Status: Compiling
Program Status: Working
*/

#include <mpi.h>
#include <stdio.h>


#include <stdlib.h>
#include <string.h>
#include <errno.h>


const int MAX_LINE_LEN = 6000;
const int MAX_WORD_LEN = 20;
const int NUM_WORDS = 1000;
const int EMBEDDING_DIMENSION= 300;
const char DELIMITER[2] = "\t";

const int COMMAND_EXIT= 0;
const int COMMAND_QUERY= 1;
const int COMMAND_CALCULATE_SIMILARITY= 2;


void distributeEmbeddings(char *filename,int world_size){
    int partLen = NUM_WORDS/(world_size-1);
    char line[MAX_LINE_LEN];
    //Reading embedding file

    FILE *file = fopen (filename, "r" );
    int wordIndex = 0;
    //Sending the necessary data to process
    float* embeddings_matrix = (float*)malloc(sizeof(float) * NUM_WORDS*EMBEDDING_DIMENSION);

    char* words = (char*)malloc(sizeof(char) * NUM_WORDS*MAX_WORD_LEN);


    for(int i = 0; i<NUM_WORDS; i++){
        fgets(line, MAX_LINE_LEN, file);

        char *word;
        word = strtok(line, DELIMITER);
```

Line 1, Column 1 Tab Size: 4 C

mpi_project.c

```c
        word = strtok(line, DELIMITER);

        strcpy(words+i*MAX_WORD_LEN, word);


        for(int embIndex = 0; embIndex<EMBEDDING_DIMENSION; embIndex++){
            char *field = strtok(NULL, DELIMITER);
            float emb = strtof(field,NULL);
            *(embeddings_matrix+i*EMBEDDING_DIMENSION+embIndex) = emb;
        }
    }
    for(int p = 1;p<=world_size-1;p++){

        char *subwords;
        subwords = (char*)malloc(sizeof(char) * partLen * MAX_WORD_LEN);

        for (int i = 0;i<partLen*MAX_WORD_LEN;i++){
            *(subwords+i) = *(words+(p-1)*MAX_WORD_LEN*partLen+i);
        }

        //Sending words to process...
        MPI_Send(
                /* data         = */ subwords,
                /* count        = */ partLen*MAX_WORD_LEN,
                /* datatype     = */ MPI_CHAR,
                /* destination  = */ p,
                /* tag          = */ 0,
                /* communicator = */ MPI_COMM_WORLD);

        //Sending embeddings to process


        float* subarray = (float*)mallocl(sizeof(float) * partLen*EMBEDDING_DIMENSION);

        for(int m = 0; m< partLen*EMBEDDING_DIMENSION ;m++){
            *(subarray+m) = *(embeddings_matrix + (p-1)*partLen*EMBEDDING_DIMENSION+m) ;
        }

        MPI_Send(
                /* data         = */ subarray,
                /* count        = */ partLen * EMBEDDING_DIMENSION,
                /* datatype     = */ MPI_FLOAT,
                /* destination  = */ p,
                /* tag          = */ 0,
                /* communicator = */ MPI_COMM_WORLD);
```

Line 77, Column 41 Tab Size: 4 C

mpi_project.c

```c
                /* communicator = */ MPI_COMM_WORLD);
    }
   //Embedding file.. has been distributed
}

float test_(char *w1, char *w2){

    float *e1 =
    float *e2 =

    float result=1;
    for (int i = 0;i<EMBEDDING_DIMENSION;i++){
        result *= *(e1+i) * *(e2+i);
    }
    return result;
}


int findWordIndex(char *words, char *query_word,int partLen){
        for(int wordIndex = 0; wordIndex<partLen; wordIndex++){
            if(strcmp((words+wordIndex*MAX_WORD_LEN), query_word)==0){
            return wordIndex;
        }
    }
    return -1;
}


int runMasterNode(int world_rank,int world_size){

    // If we are rank 0, set the number to -1 and send it to process 1
    int partLen = NUM_WORDS / world_size-1;
    int slaveSize = world_size-1;
    distributeEmbeddings("./word_embeddings.txt",world_size);

    while(1==1){
        printf("Please type a query word:\n");

        char queryWord[256];
        scanf( "%s" , queryWord);
        printf("Query word:%s\n",queryWord);
        char exit1[256] = "EXIT";
        char exit2[256] = "exit";

        if(strcmp(exit1,queryWord)==0 || strcmp(exit2,queryWord)==0){ //EXİT gönder
```

Line 96, Column 17 Tab Size: 4 C

mpi_project.c

```c
        if(strcmp(exit1,queryWord)==0 || strcmp(exit2,queryWord)==0){ //EXIT gönder

            for (int p = 1;p<=world_size-1;p++){
                MPI_Send(
                  /* data        = */ (void*)&COMMAND_EXIT,
                  /* count       = */ 1,
                  /* datatype    = */ MPI_INT,
                  /* destination = */ p,
                  /* tag         = */ 0,
                  /* communicator = */ MPI_COMM_WORLD);
            }
            return 0;
        }else{

            for(int p = 1; p<=slaveSize;p++){
                //Command is being sent to process
                MPI_Send( (void *)&COMMAND_QUERY,  1, MPI_INT, p,0, MPI_COMM_WORLD);
                MPI_Send( queryWord,MAX_WORD_LEN,MPI_CHAR,p,0,MPI_COMM_WORLD);
                //Query is sent to process
            }

            float *query_embeddings =  (float*)malloc(sizeof(float)*EMBEDDING_DIMENSION);

            int found =0;

            for (int p = 1;p<=slaveSize;p++){
                int *target_word_index =  (int*)malloc(sizeof(int));
                MPI_Recv(target_word_index, 1,MPI_FLOAT,p,0,MPI_COMM_WORLD,MPI_STATUS_IGNORE);

                if (*target_word_index >= 0){
                    MPI_Recv(query_embeddings,EMBEDDING_DIMENSION,MPI_FLOAT,p,0,MPI_COMM_WORLD,MPI_STATUS_IGNORE);

                    //query command is received

                    found = 1;
                }
            }

            if (!found){
                printf("Query word was not found\n");
                continue;
            }
            for (int p=1;p<=slaveSize;p++){
                MPI_Send((void *)&COMMAND_CALCULATE_SIMILARITY,1,MPI_INT,p,0,MPI_COMM_WORLD);

```

Line 158, Column 1 Tab Size: 4 C

mpi_project.c

```c
                //COMMAND_CALCULATE_SIMILARITY is sent to process

                MPI_Send(query_embeddings,EMBEDDING_DIMENSION,MPI_FLOAT,p,0,MPI_COMM_WORLD);

            }

            float *bestPscore = (float*)malloc(sizeof(float) * (world_size-1)*(world_size-1));
            char* words = (char*)malloc(sizeof(char) * MAX_WORD_LEN*(world_size-1)*(world_size-1));

            for(int p=1;p<=slaveSize;p++){

                MPI_Recv(bestPscore+(p-1)*slaveSize, slaveSize, MPI_FLOAT,p,0,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
                MPI_Recv(words+(p-1)*slaveSize*MAX_WORD_LEN, MAX_WORD_LEN*slaveSize, MPI_CHAR,p,0,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
                //Best P Scores and Words are received
            }

            //ALL subarrays are received BY MASTER NODE

            int ptrScore[world_size-1]; //pointer array for maximum of subarrays received by slaves

            float *outputPscore = (float*)malloc(sizeof(float) * slaveSize);
            char *outputWords = (char*)malloc(sizeof(char) * MAX_WORD_LEN*slaveSize);

            //initilization ptrScore is 0-10-20 ... 90. It holds the starting point of subarrays.
            for(int i = 0;i<slaveSize;i++){
                ptrScore[i] = i*slaveSize;
            }

            //starting of finding part of most relevant p words
            int maxIndex;//it is the index of next maximum relevant word
            for(int j =0;j<slaveSize;j++){
                float maxSimilarityScore = -1;

                for(int i = 0;i<slaveSize;i++){
                    if(*(bestPscore + ptrScore[i])>maxSimilarityScore){
                        maxSimilarityScore = *(bestPscore + ptrScore[i]);
                        maxIndex = i;
                    }
                }
                *(outputPscore + j) = *(bestPscore + ptrScore[maxIndex]);

                for(int i = 0;i<MAX_WORD_LEN;i++){
                    *(outputWords + j*MAX_WORD_LEN + i) = *(words + ptrScore[maxIndex] * MAX_WORD_LEN+i);
```

Line 194, Column 1 Tab Size: 4 C

mpi_project.c

```c
                *(outputWords + j*MAX_WORD_LEN + i) = *(words + ptrScore[maxIndex] * MAX_WORD_LEN+i);
            }

            ptrScore[maxIndex]++;
        }
        //end of finding most relevant words

        //printing result
        printf("TOP %d RESULTS:\n",slaveSize);
        for(int i= 0;i<world_size-1;i++){
            int c = i * MAX_WORD_LEN;
            while (*(outputWords+c) != '\0') {
                printf("%c", outputWords[c]);
                c++;
            }
            printf(": %f\n", *(outputPscore+i));
        }

    }

  }
  return 0;
}

int runSlaveNode(int world_rank,int world_size){

    int partLen = NUM_WORDS / (world_size-1);
    int slaveSize = world_size-1;
    //Receiving words

    char* words = (char*)malloc(sizeof(char) * MAX_WORD_LEN*partLen);

    MPI_Recv(words, partLen*MAX_WORD_LEN, MPI_CHAR, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

    //Words received

    float* embeddings_matrix = (float*)malloc(sizeof(float) * partLen*EMBEDDING_DIMENSION);

    //Process started to receive embedding part

    MPI_Recv(embeddings_matrix, partLen*EMBEDDING_DIMENSION, MPI_FLOAT, 0, 0, MPI_COMM_WORLD,MPI_STATUS_IGNORE);

    //Process received embedding part
```

Line 235, Column 33 Tab Size: 4 C

mpi_project.c

```c
    while(1==1){

        //Process is waiting for command

        int command;

        MPI_Recv(
              /* data         = */ &command,
              /* count        = */ 1,
              /* datatype     = */ MPI_INT,
              /* source       = */ 0,
              /* tag          = */ 0,
              /* communicator = */ MPI_COMM_WORLD,
              /* status       = */ MPI_STATUS_IGNORE);

        //Command received

        if(command == COMMAND_EXIT){
            return 0;
        }else if(command == COMMAND_QUERY){
            //printf("command=1 \n");
            char* query_word = (char*)malloc(sizeof(char) * MAX_WORD_LEN);

            MPI_Recv(
              /* data         = */ query_word,
              /* count        = */ MAX_WORD_LEN,
              /* datatype     = */ MPI_CHAR,
              /* source       = */ 0,
              /* tag          = */ 0,
              /* communicator = */ MPI_COMM_WORLD,
              /* status       = */ MPI_STATUS_IGNORE);

            //Query word received

            int wordIndex = findWordIndex(words, query_word,partLen);


            float* query_embeddings = (embeddings_matrix+wordIndex*EMBEDDING_DIMENSION);

            int* WordIndexInMasterNode = (int*)malloc(sizeof(int));
            *WordIndexInMasterNode = wordIndex*world_rank;

            MPI_Send(WordIndexInMasterNode, 1,MPI_INT,0,0,MPI_COMM_WORLD);
```

Line 294, Column 1 Tab Size: 4 C

mpi_project.c

```c
            if(*WordIndexInMasterNode >=0 ){
                MPI_Send(query_embeddings, EMBEDDING_DIMENSION, MPI_FLOAT,0,0,MPI_COMM_WORLD);
            }

        }else if(command == COMMAND_CALCULATE_SIMILARITY){


            float* query_embeddings = (float*)malloc(sizeof(float) *EMBEDDING_DIMENSION);

            MPI_Recv(query_embeddings,EMBEDDING_DIMENSION,MPI_FLOAT,0,0,MPI_COMM_WORLD,MPI_STATUS_IGNORE);


            //Calculating similarities by using a way similar to insertion sort. It finds most relevant P words among partLen words.

            int mostSimilarWordIndex = -1;
            float maxSimilarityScore = -1;

            float* similarityScores = (float*)malloc(sizeof(float) * partLen);

            int *bestPindex = (int*)malloc(sizeof(int) * world_size-1); //best P words index
            float *bestPscore = (float*)malloc(sizeof(float) * world_size-1);//best P words scores


            //initilization

            for(int i = 0;i<world_size-1;i++){
                *(bestPscore+i) = -1;
                *(bestPindex+i) = -1;
            }


            int last=-1;//it represent whether the array which has length P is full or not

            for(int wordIndex = 0; wordIndex<partLen; wordIndex++){
                float similarity = 0.0;

                for(int embIndex = 0; embIndex<EMBEDDING_DIMENSION; embIndex++){
                    float emb1 = *(query_embeddings + embIndex);
                    float emb2 = *(embeddings_matrix + wordIndex*EMBEDDING_DIMENSION + embIndex);
                    similarity +=(emb1*emb2);
                }

                *(similarityScores + wordIndex) = similarity;
```

Line 338, Column 1 Tab Size: 4 C

mpi_project.c

```c
                if(similarity>maxSimilarityScore){
                    mostSimilarWordIndex = wordIndex;
                    maxSimilarityScore=similarity;
                }
                int position;


                // if bestPscore and bestPindex is full and new element should be inside of array,
                // we need to be careful the last element of bestPscore and bestPindex when shifting arrays

                if (last < world_size-2){ // if array is not full..
                    position = last;
                    last++;

                    while ( position >=0 && similarity > *(bestPscore+position) ){
                        *(bestPscore + position + 1) = *(bestPscore + position);
                        *(bestPindex + position + 1) = *(bestPindex + position);
                        position--;
                    }

                }else{ // if the array is full, search for a right place to insert by starting last. It is like insertion sort.
                    position = world_size-2;

                    while ( position >=0 && similarity > *(bestPscore+position) ){
                        if(position != world_size-2){ // carefuling for last element of arrays
                            *(bestPscore + position + 1) = *(bestPscore + position);
                            *(bestPindex + position + 1) = *(bestPindex + position);
                        }
                        position--;
                    }

                }

                // if position == world_size-2, then new word's score is less than all words' scores in the bestPscore

                if(position != world_size-2){//insert new words and scores to the appropriate position which is position + 1, it is the same logic in inse
                    *(bestPscore + position+1) = similarity;
                    *(bestPindex + position+1) = wordIndex;
                }

            }
```

Line 368, Column 17 Tab Size: 4 C

mpi_project.c

```c
            char* sortedwords = (char*)malloc(sizeof(char) * MAX_WORD_LEN*world_size-1);

            for(int i=0;i<world_size-1;i++){
                int nextIndex = *(bestPindex+i);
                for(int j = 0; j< MAX_WORD_LEN; j++){
                    *(sortedwords+i*MAX_WORD_LEN +j) = *(words + (nextIndex * MAX_WORD_LEN) +j);
                }
            }

            //send the bestPscore and sortedwords to master node

            MPI_Send(bestPscore, world_size-1, MPI_FLOAT,0,0,MPI_COMM_WORLD);
            MPI_Send(sortedwords, (world_size-1) * MAX_WORD_LEN, MPI_CHAR,0,0,MPI_COMM_WORLD);

        }


    }

    return 0 ;
}



int main(int argc, char** argv) {


    // Initialize the MPI environment
    MPI_Init(NULL, NULL);

    // Get the number of processes
    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

    // Get the rank of the process
    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

    // Get the name of the processor
    char processor_name[MPI_MAX_PROCESSOR_NAME];
    int name_len;
    MPI_Get_processor_name(processor_name, &name_len);

    // Print off a hello world message
```

Line 412, Column 10 Tab Size: 4 C

mpi_project.c

```c


    // Initialize the MPI environment
    MPI_Init(NULL, NULL);

    // Get the number of processes
    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

    // Get the rank of the process
    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

    // Get the name of the processor
    char processor_name[MPI_MAX_PROCESSOR_NAME];
    int name_len;
    MPI_Get_processor_name(processor_name, &name_len);

    // Print off a hello world message

    // We are assuming at least 2 processes for this task
    if (world_size < 2) {
        fprintf(stderr, "World size must be greater than 1 for %s\n", argv[0]);
        MPI_Abort(MPI_COMM_WORLD, 1);
    }

    int wordIndex;

    if (world_rank == 0) {

        runMasterNode(world_rank,world_size);

    }else {

        runSlaveNode(world_rank,world_size);

    }

    MPI_Finalize();

    //Processors stopped
}


```

Line 467, Column 1 Tab Size: 4 C